1. Principe de la technique algorithmique « diviser pour régner »

- **Diviser :** découper un problème initial en sous-problème (souvent en deux parties plus ou moins égales)
- **Régner :** résoudre les sous problèmes (généralement récursivement)
- **Combiner :** calculer une solution au problème initial à partir des solutions des sous-problèmes

Cette technique permet souvent d'obtenir des algorithmes d'une meilleure complexité.

2. Exemple : la **recherche dichotomique**

Version itérative :

On rappelle que la recherche dichotomique consiste vérifier la présence d'une valeur dans un ensemble de valeurs déjà triées. On compare la valeur au milieu de l'ensemble avec celle cherchée. Si elles correspondent on retourne son indice. Dans le cas contraire on change les bornes de recherche avec celles du demi-ensemble des valeurs plus petites ou plus grandes en fonction de la comparaison. On recommence ainsi tant que la borne droite de l'ensemble de recherche est supérieure ou égale à celle de la borne gauche.
Dans le cas où la valeur n'est pas dans l'ensemble on retourne -1

Une implémentation en python peut être :

```
def dicho_ite(T,valeur) :
    gauche = 0
    droite = len(T)-1
    while droite >= gauche :
        milieu = (droite + gauche)//2
        if T[milieu] == valeur :
            trouve = True
            return milieu
        elif valeur > T[milieu]:
            gauche = milieu+1
        else :
            droite = milieu -1
    return -1
```

Pour s'approprier cette technique, on s'intéresse à la valeurs des variables lors de l'exécution de :

```
dicho_ite([-5,2,9,12,88] , 15)
```

valeur	*gauche*	*droite*	*milieu*	*T[milieu]*
15	0	4	2	9
15	3	4	3	12
15	4	4	4	88
15	4	3		

Valeur retournée : -1

La boucle *while* se termine avec la condition droite >= gauche qui n'est plus vraie. Ainsi la valeur -1 est retournée

Version récursive :

Compléter l'implémentation en python d'une version récursive :

```
def dicho_rec(T , valeur , gauche = 0 , droite = -1) :
    if droite == -1 : droite = len(T)-1
    if gauche > droite : return -1
    milieu = (droite + gauche)//2
    if T[milieu] == valeur :
        return valeur
    elif valeur > T[milieu]:
        return dicho_rec(T , valeur , milieu +1 , droite)
    else :
        return dicho_rec(T , valeur , gauche , milieu-1)
```

Pour s'approprier cette fonction, on s'intéresse à la valeur des variables lors de l'exécution de :

```
dicho_rec([-5,2,9,12,88] , 15)
```

dicho_rec ([-5,2,9,12,88] , 15

gauche	*droite*	*milieu*	*T[milieu]*
0	*4*	*2*	*9*

return dicho_rec ([-5,2,9,12,88] , 15 , 3 , 4)

gauche	*droite*	*milieu*	*T[milieu]*
3	*4*	*3*	*12*

return dicho_rec ([-5,2,9,12,88] , 15 , 4 , 4)

gauche	*droite*	*milieu*	*T[milieu]*
4	*4*	*2*	*88*

return dicho_rec ([-5,2,9,12,88] , 15 , 4 , 3)

gauche	*droite*	*milieu*	*T[milieu]*
4	*3*		

return *-1*

Valeur retournée : -1

3. Exemple : le **tri fusion**

Après les tris par insertion et par sélection, qui permettent de trier un tableau de n nombres en $O(n^2)$. La méthode « diviser pour régner » permet de décrire un algorithme de tri de meilleure complexité, à savoir $O(n \log n)$ pour un tableau de taille n . Le principe de l'algorithme suit les phases de la méthode « diviser pour régner ».

1. **Diviser** : découpe du tableau initial en deux sous-tableaux de taille (environ) égale.

2. **Régner** : Résoudre tri des deux sous-tableaux, grâce à deux appels récursifs.

3. **Combiner** : fusion des deux sous-tableaux triés pour produire le tableau trié complet.

Appliquer la méthode à la liste suivante :

<table>
<tr><td></td><td colspan="9">[8,2,5,4,9,6,1,7,3]</td></tr>
<tr><td>Diviser</td><td colspan="4">[8,2,5,4]</td><td colspan="5">[9,6,1,7,3]</td></tr>
<tr><td>Diviser</td><td colspan="2">[8,2]</td><td colspan="2">[5,4]</td><td colspan="2">[9,6]</td><td colspan="3">[1,7,3]</td></tr>
<tr><td rowspan="2">Diviser</td><td rowspan="2">[8]</td><td rowspan="2">[2]</td><td rowspan="2">[5]</td><td rowspan="2">[4]</td><td rowspan="2">[9]</td><td rowspan="2">[6]</td><td rowspan="3">[1]</td><td colspan="2">[7,3]</td></tr>
<tr><td>[7]</td><td>[3]</td></tr>
<tr><td>Fusionner</td><td colspan="2">[2,8]</td><td colspan="2">[4,5]</td><td colspan="2">[6,9]</td><td colspan="2">[3,7]</td></tr>
<tr><td>Fusionner</td><td colspan="2">[2,8]</td><td colspan="2">[4,5]</td><td colspan="2">[6,9]</td><td colspan="3">[1,3,7]</td></tr>
<tr><td>Fusionner</td><td colspan="4">[2,4,5,8]</td><td colspan="5">[1,3,6,7,9]</td></tr>
<tr><td>Fusionner</td><td colspan="9">[1,2,3,4,5,6,7,8,9]</td></tr>
</table>

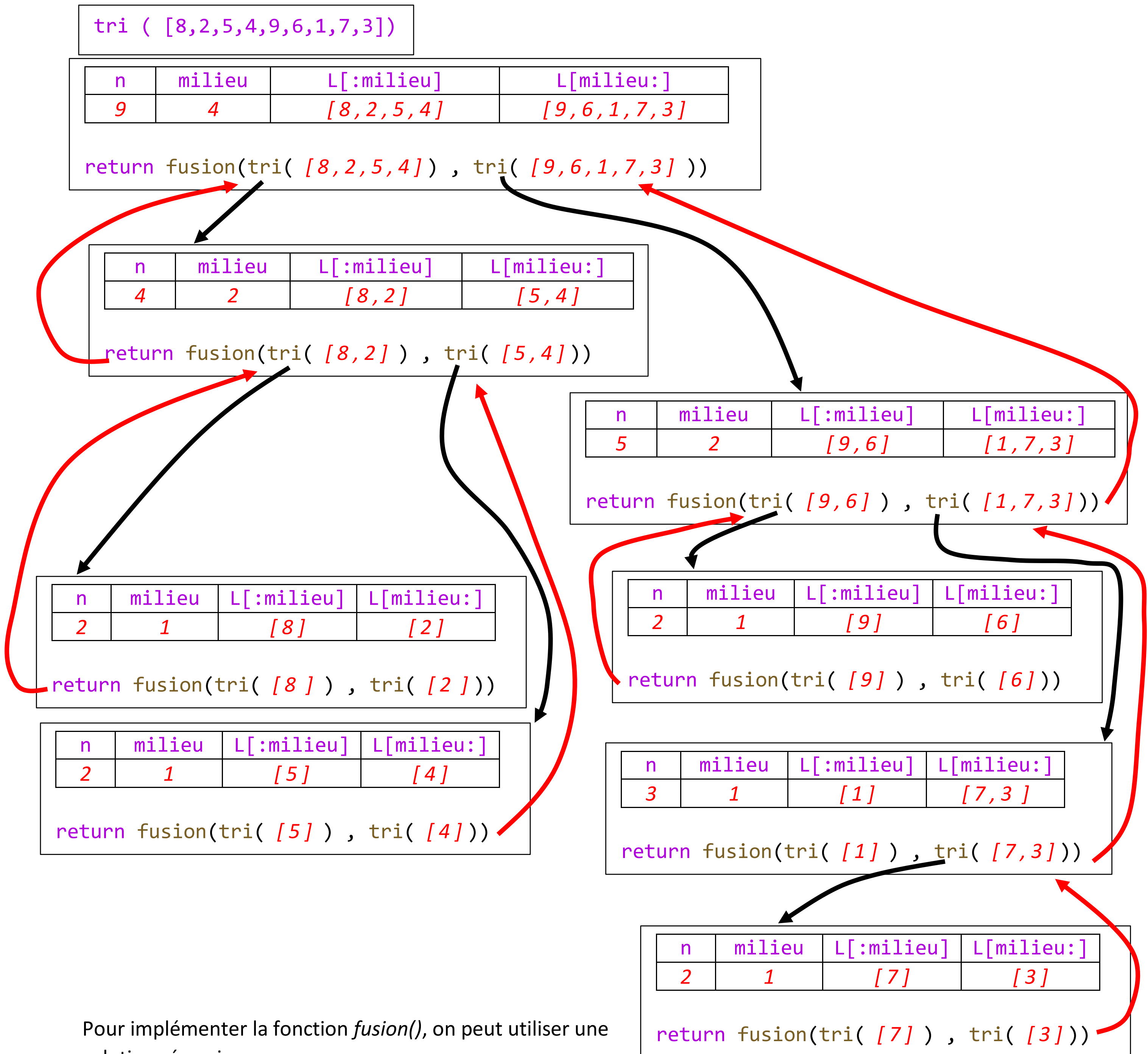
Une implémentation en python peut être la suivante :

```python
def tri(L) :
    n = len(L)
    if n < 2 :
        return L
    else :
        milieu = n // 2
        return fusion(tri(L[:milieu]) , tri(L[milieu:]))
```

La fonction *fusion()* sera implémentée dans la suite. Elle prend en argument 2 listes et retourne une liste qui comprend les éléments triés de ces 2 listes.

Pour s'approprier cette fonction, on s'intéresse à la valeur des variables lors de l'exécution de :

```
tri([8,2,5,4,9,6,1,7,3])
```



Pour implémenter la fonction *fusion()*, on peut utiliser une solution récursive :

```python
def fusion(L1 , L2) :
    assert type(L1) == list , "L'argument L1 doit être une liste"
    assert type(L2) == list , "L'argument L2 doit être une liste"
    if L1 == [] : return L2
    elif L2 == [] : return L1
    elif L1[0] <  L2[0] :
        return [L1[0]] + fusion(L1[1:] , L2)
    else :
        return [L2[0]] + fusion(L1 , L2[1:])
```

On peut également utiliser une méthode itérative :

```
def fusion(L1 , L2) :
    i1 , i2 = 0 , 0
    L = []
    while i1 < len(L1) and i2 < len(L2) :
        if L1[i1] < L2[i2] :
            L.append(L1[i1])
            i1 += 1
        else :
            L.append(L2[i2])
            i2 += 1

    if i1 < len(L1) :
        return L + L1[i1:]

    if i2 < len(L2) :
        return L + L2[i2:]
```

La complexité des tris par sélection ou insertion sont en $O(n^2)$.

<u>Analyse de la complexité du tri fusion :</u>

⇨ Pour réaliser un tri fusion sur une liste de taille $n = 16 = 2^4$, on divise 4 fois la liste par 2 :

[x , x , x , x , x , x , x , x , x , x , x , x , x , x , x , x]
1ère division : [x , x , x , x , x , x , x , x] + [x , x , x , x , x , x , x , x]
2ième division : [x , x , x , x] + [x , x , x , x] + [x , x , x , x] + [x , x , x , x]
3ième division : [x , x] + [x , x] + [x , x] + [x , x] + [x , x] + [x , x] + [x , x] + [x , x]
4ième division : [x] + [x] +[x] + [x] +[x] + [x] +[x] + [x] +[x] + [x] +[x] + [x] +[x] + [x] +[x] + [x]

Pour trouver ce nombre de division, on peut utiliser la fonction mathématique $log_2()$ définie par $log_2(2^x) = x$. On a ainsi : $log_2(16) = log_2(2^4) = 4$
Sur calculatrice, cette fonction n'est pas toujours disponible. On utilise alors le fait que $log_2(x) = \frac{\ln(x)}{\ln(2)}$.

<u>Questions</u> :

- Calculer le nombre de division pour une liste de taille $n = 64$:

$$log_2(64) = log_2(2^6) = 6 \quad \text{ou} \quad log_2(64) = \frac{\ln(64)}{\ln(2)} = 6$$

- Calculer le nombre de division pour une liste de taille $n = 1\,048\,576 = 2^{20}$:

$$log_2(1\,048\,576) = log_2(2^{20}) = 20 \quad \text{ou} \quad log_2(1\,048\,576) = \frac{\ln(1\,048\,576)}{\ln(2)} = 20$$

- Calculer le nombre de division pour une liste de taille $n = 1\,073\,741\,824 = 2^{30}$:

$$log_2(1\,073\,741\,824) = log_2(2^{30}) = 20 \quad \text{ou} \quad log_2(1\,073\,741\,824) = \frac{\ln(1\,073\,741\,824)}{\ln(2)} = 30$$

⇨ Pour réaliser un tri fusion sur une liste de taille n on réalise ainsi $log_2(n)$ divisions. Pour chacune d'elles, on a environ n opérations, car il s'agit de fusionner les n éléments de la liste. On peut ainsi dire que la complexité de cet algorithme de tri fusion est en $O(n \times log_2(n))$.

4. Exemple : le **tri rapide** (quick sort)

Cette méthode de tri consiste à placer un élément de la liste, appelé pivot, à sa place définitive en permutant tous les éléments de telle sorte que tous ceux qui sont inférieurs au pivot soient à sa gauche et que tous ceux qui sont supérieurs soient à sa droite. Pour chacune des sous-listes gauche et droite obtenues, on répète alors le processus : on y définit un nouveau pivot et on répartit les éléments autour de celui-ci. Le processus est répété récursivement jusqu'à ce que l'ensemble des éléments soient triés.

La position du pivot peut être définit aléatoirement.

Dans cette méthode :

1. **Diviser** : on divise le problème (liste gauche et liste droite autour d'un pivot)

2. **Régner** : On trie les valeurs des sous-listes en répartissant les éléments autour du pivot

3. **Combiner** : On reconstitue la liste entière en ajoutant chaque morceau trié

Pour l'illustrer, on utilise cette méthode sur la liste suivante :

	[8 , 2 , 5 , 4 , 9 , 6 , 1 , 7 , 3]
Etape 1	[2 , 5 , 4 , 1 , 3] + [6] + [8 , 9 , 7]
Etape 2	[2 , 4 , 1 , 3] + [5] + [ ] + [6] + [ ] + [7] + [8 , 9]
Etape 3	[1] + [2] + [4 , 3] + [5] + [6] + [7] + [ ] + [8] + [9]
Etape 4	[1] + [2] + [3] + [4] + [ ] + [5] + [6] + [7] + [ ] + [8] + [9]
	[1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9]

Une implémentation en python peut être le suivante :

```
def tri_rapide(L) :
    assert type(L) == list , "L'argument doit être une liste"
    if len(L) < 2 : return L
    pivot = randint(0,len(L)-1)
    L1 = []
    L2 = []
    for i in range(len(L)) :
        if L[i] < L[pivot] : L1.append(L[i])
        elif L[i] > L[pivot] : L2.append(L[i])
        elif i != pivot : L1.append(L[i]) # pour les doublons
    return tri_rapide(L1) + [L[pivot]] + tri_rapide(L2)
```